Carnegie Mellon
Software Engineering Institute

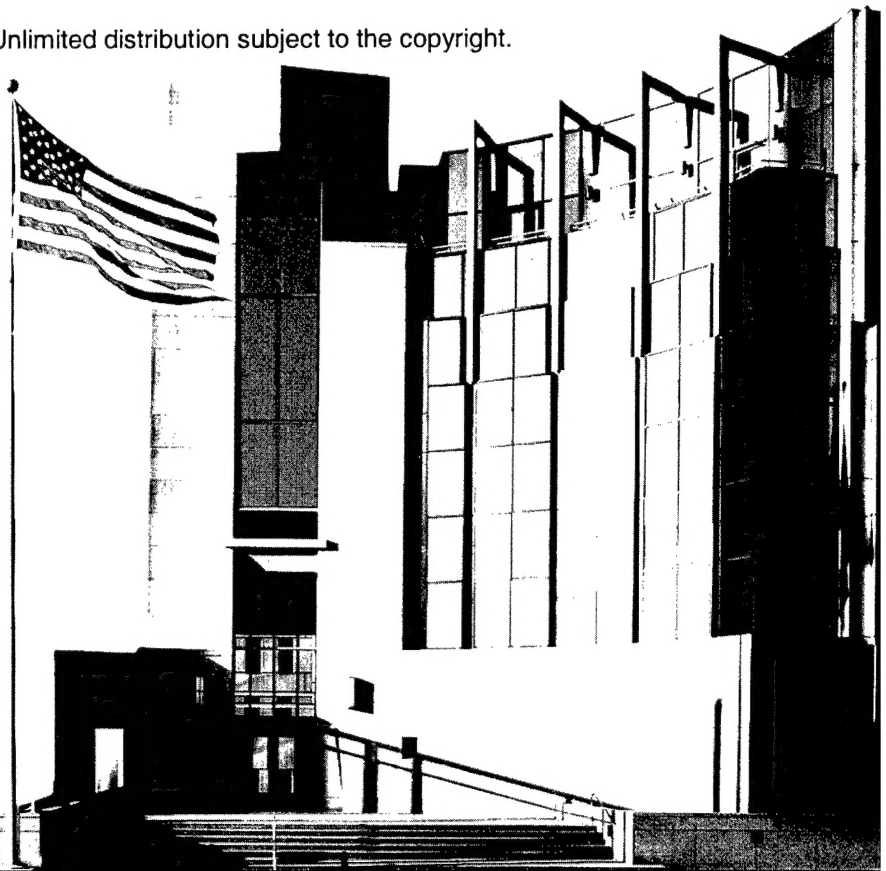
Beyond the Black Box: A Case Study in C to Java Conversion and Product Extensibility

Pisey Huy
Grace A. Lewis
Ming-hsun Liu

August 2001

COTS-Based Systems Initiative

Technical Note
CMU/SEI-2001-TN-017



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Beyond the Black Box: A Case Study in C to Java Conversion and Product Extensibility

Pisey Huy
Grace A. Lewis
Ming-hsun Liu

August 2001

COTS-Based Systems Initiative

Technical Note
CMU/SEI-2001-TN-017

Contents

Abstract		**vii**
1	**Introduction**	**1**
2	**NDBS 1.0**	**2**
	2.1 Architecture of NDBS 1.0	2
	2.2 Users of NDBS 1.0	4
3	**NDBS 2.0**	**5**
	3.1 Development of NDBS 2.0	5
	3.2 Architecture of NDBS 2.0	17
4	**Extending the Functionality of NDBS 2.0**	**21**
5	**Summary**	**27**
References		**29**
Appendix A:	**Acronyms and Terms**	**31**
Appendix B:	**Microsoft Security Architecture**	**33**
Appendix C:	**NDBS 2.0 UML Class Diagrams**	**34**

List of Figures

Figure 1: NDBS 1.0 Architecture 3

Figure 2: High-Level UML Class Diagram for Netscape Keystore Management 8

Figure 3: High-Level UML Class Diagram for Cryptography Routines 10

Figure 4: High-Level UML Class Diagram for the Berkeley DB 1.85 Hash Structure 12

Figure 5: High-Level UML Class Diagram for the Keystore Service Provider Interface 13

Figure 6: PSM Architecture 15

Figure 7: Runtime Architectural Diagram for NDBS 2.0 17

Figure 8: NDBS 2.0 Architecture 18

Figure 9: Mapping of Components to the NDBS 2.0 Architecture 19

Figure 10: Bridge Pattern in NDBS 2.0 21

Figure 11: Code in NDBSKeystore.java that Implements the Bridge Pattern 22

Figure 12: Façade Pattern in NDBS 2.0 for HMAC SHA-1 24

Figure 13: Java Sample Code for Implementing the Façade Pattern for the HMAC with SHA-1 Algorithm in NDBS (Part 1 of 2) 25

Figure 14: Java Sample Code for Implementing the Façade Pattern for the HMAC with SHA-1 Algorithm in NDBS (Part 2 of 2) 26

Figure 15: Microsoft Security Architecture 33

Figure 16: UML Class Diagram for the Keystore Management Layer 34

Figure 17: UML Class Diagram for the Netscape Keystore Layer 35

Figure 18: UML Class Diagram for the Netscape Crypto Façade Layer 36

Figure 19: UML Class Diagram for the Berkeley DB Access Layer 37

List of Tables

Table 1:	Lines of Code in NDBS 1.0 and NDBS 2.0	6
Table 2:	Lines of Code in NDBS 2.0 After Adding Write and Delete Capabilities	6

Abstract

This case study describes the experience of converting and enhancing NDBS 1.0 (Netscape Database Keystore), a programmatic library to extract private keys and digital certificates from a Netscape database written in C and Java. The result of this work is NDBS 2.0, a 100% Java version of NDBS 1.0 designed to support other keystores easily. NDBS 2.0 also includes write and delete capabilities, features that were not present in NDBS 1.0. The case study describes the experience of the conversion and development process, difficulties, and lessons learned.

1 Introduction

In 1999, an SEI project required the capability to programmatically extract private keys and digital certificates from a Netscape Communicator v4.5 database. The result of this effort is a programmatic library named Netscape Database Keystore (NDBS 1.0) and an SEI technical note describing the experience [Plakosh 99]. The limitations of NDBS 1.0 are

- platform dependency because it is written in Java and interfaces with C code using the Java Native Interface (JNI)
- inability to write or delete certificates and private keys from the Netscape database (reading is enabled)

NDBS 2.0 is a platform-independent, 100% Java version of NDBS 1.0, with added functionality to write and delete certificates and private keys. It is designed for extensibility so that future projects may extend NDBS to support other keystores such as Microsoft Cryptographic Service Provider (CSP).

This case study describes

1. converting NDBS 1.0 and part of the underlying database manager from C to Java
2. using design patterns for extensibility
3. adding write and delete capabilities

Following the introduction, we describe the features and architecture of NDBS 1.0 in Section 2. In Section 3, we describe the features and architecture of NDBS 2.0, along with the conversion experience and the addition of write and delete capabilities. In Section 4, we present the use of design patterns to provide an extensible architecture. Finally, the lessons learned are included in Section 5.

2 NDBS 1.0

The Netscape Database Keystore (NDBS) 1.0 is a programmatic library for extracting private keys and X.509v3 certificates from Netscape products. Since Netscape's Network Security Services (NSS) would not divulge private key management information or specifications, NDBS 1.0 was built so that Public Key Infrastructure (PKI)-enabled products could access private keys and certificates from Netscape's database. NDBS 1.0 is a Java Cryptography Extension (JCE) 1.2-compliant provider for keystore services designed to read certificates and private key material from the Netscape database. Writing and deleting certificates and private keys were not supported in this version.

NDBS 1.0 features include a 100% Java Cryptography Extension (JCE)-1.2 compliant Keystore Service Provider Interface (SPI). Its capabilities include the following:

- maps the internal files where Netscape stores its security information (key3.db and cert7.db) to one logical database
- reads password protected or unprotected Netscape database files
- works with Java Development Kit (JDK) 1.2 keytool key/certificate management tool
- supports certificate chains
- supports Windows NT, Windows 95/98/2000, and Solaris 2.5.1 and higher

For further information on NDBS 1.0 refer to the SEI technical note *Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software* [Plakosh 99] and to the URL <http://agora.sei.cmu.edu/ndbs/>.

2.1 Architecture of NDBS 1.0

The architecture of NDBS 1.0 is a layered system, as shown in Figure 1. The layers of the architecture are described in Figure 1.

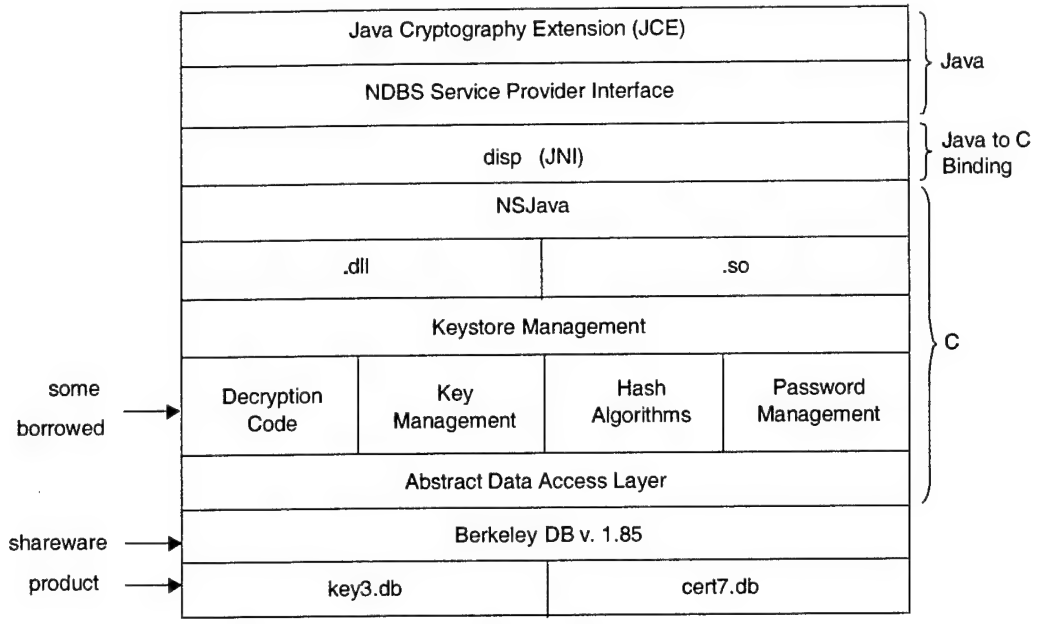


Figure 1: NDBS 1.0 Architecture

- JCE - Java Cryptography Extension: JCE is a package that provides a framework and implementations for encryption, key generation, key agreement, and Message Authentication Code (MAC) algorithms. It is designed so that other qualified cryptography libraries can be plugged in as service providers, and new algorithms can be added.
- NDBS Service Provider Interface: This layer acts as the SPI. The use of this provider is specified in the java.security file.
- disp (JNI): The Java Native Interface is a standard programming interface for writing Java native methods and embedding the Java Virtual Machine into native applications. The primary goal is binary compatibility of native method libraries across all Java Virtual Machine implementations on a given platform. This layer is used to enable the NDBS SPI to communicate with routines written in the C programming language.
- NSJava: NSJava is the library that contains the calls to the routines that are able to extract, decrypt, and decode information from the Netscape database files.
- .dll and .so: Because the output from the C programming language compiler is platform dependent, .dll is the NSJava version for the Windows platform, and .so is the NSJava version for the Solaris platform.
- Keystore Management: This layer contains the routines that extract, decrypt, and decode information from the Netscape database.
- Decryption code, key management, hash algorithms, password management: These are specific routines for performing the described functions. Some of this code was downloaded from several sources found on the Internet.
- Abstract data access layer: This is the C Application Program Interface (API) for the Berkeley database (DB).

- Berkeley DB v1.85: This is the Database Management System used by the Netscape browser to manage its keystores.
- key3.db: This is the Netscape private key database that stores the encrypted private key information.
- cert7.db: This is the Netscape certificate database that stores X.509v3 certificates, as well as links to the key3.db file.

2.2 Users of NDBS 1.0

The users of NDBS 1.0 are Java developers who need programmatic access to a Netscape keystore database (certificates and private keys). It is available only for Microsoft Windows under the Intel architecture and Solaris platforms because it uses JNI to interface between Java code and platform-dependent C code. NDBS 1.0 was made available to the public in August 1999. From August 1999 to March 2001 there have been 4,546 downloads of the abstract, 3,226 downloads of the PDF version of the report, and 320 downloads of the actual software. Several requests for the NDBS 1.0 source have been made from universities, software consulting and development companies, and communication companies.

The success of NDBS 1.0 is its ability to read password protected or unprotected Netscape database files.

The limitations of NDBS 1.0 are that it

- is only proven to operate on Solaris platforms and Microsoft Windows platforms under the Intel architecture. The reason for this is that NDBS 1.0 is partly written in C and therefore is platform dependent.
- cannot write or update the Netscape database files
- cannot delete key materials or certificates
- only operates on Netscape database files
- does not have an automated installation process

These limitations drove the development of NDBS 2.0.

3 NDBS 2.0

NDBS 2.0 is a 100% Java version of NDBS 1.0 with the addition of write and delete capabilities so that users can read, write, and delete certificates and private keys from a Netscape keystore.

The users of NDBS 2.0 are the same users of NDBS 1.0; that is, Java developers who need programmatic access to a Netscape keystore database. The difference is that NDBS 2.0 is platform-independent because it is 100% Java. Installation requirements are

- Java 2 (JDK 1.2 or later version)
- RSA BSAFE Crypto-J 2.1 or Entrust/Toolkit Java™ Edition 4.1 or equivalent security provider, conforming to JCE 1.2 SPI for Crypto
- JCE 1.2 compliant crypto provider that supports Triple Data Encryption Standard (DES) with Cipher Block Chaining (CBC) and Standard Block Padding, and Keyed-Hash Message Authentication Code (HMAC) with SHA-1 algorithms, such as SunJCE, IAIK, JCSI, or Crypto-J.

3.1 Development of NDBS 2.0

The code for providing 100% Java-equivalent functionality of NDBS 1.0 was converted completely. We did not, however, convert the crypto routines because we used classes provided either by Java or by a JCE 1.2-compliant crypto provider. Both systems are layered. The LOC[1] for each system and layer are listed in the table on the next page.

[1] Lines of C code were counted using a tool called CodeCount from the University of Southern California. This tool counts logical LOCs classified as compiler directives, data lines, or executable lines. It excludes comments (whole or embedded) and blank lines (http://sunset.usc.edu/research/CODECOUNT/license.html).
Lines of Java code were counted using a tool called JavaCount from the University of Hawaii. This tool counts non-comment lines of Java source code (http://csdl.ics.hawaii.edu/Tools/JavaCount/JavaCount.html).

Table 1: Lines of Code in NDBS 1.0 and NDBS 2.0

	NDBS 1.0			NDBS 2.0	
Development Stage	**C LOC**	**Java LOC**	**Program Files/Classes**	**Java LOC**	**Classes**
Berkeley DB Hash Functions	1200	0	6	1621	8
Cryptography Routines	910	0	4	212[1]	3
Netscape Key Store	788	0	1	1323	8
NDBS Key Store SPI	0	499[2]	6	214	3
SUB-TOTAL	2898	499	11/6	1749	22
TOTAL		**3397**		**3370**	

After adding the write and delete capabilities, lines of code (LOC) count for NDBS 2.0 changed as shown below.

Table 2: Lines of Code in NDBS 2.0 After Adding Write and Delete Capabilities

	NDBS 2.0 Java LOC Count	
Development Stage	**Before Write/Delete**	**After Write/Delete**
Berkeley DB Hash Functions	1621	3289
Cryptography Routines	212	215
Netscape Key Store	1323	2535
NDBS Key Store SPI	214	563
TOTAL	**3370**	**6602**

The major challenges in the conversion are listed below and detailed in the remaining sections.

- The code used by *NDBSKeystore* in NDBS 1.0 is written in C - source code downloaded from the Internet, as well as code developed at the SEI. Because the motivation in NDBS 1.0 was to prove that access to the Netscape keystores was possible, there is no documentation, except for the information in [Plakosh 99] and the references listed in this technical note.
- Converting from a structured language like C to an object-oriented language like Java required a complete redesign of the system. Instead of designing around functions, NDBS 2.0 was designed around data structures. These were merged into classes with the functions that operate on them, following the object-oriented concept of encapsulation.
- To make use of Java exception handling, all error conditions and error reporting had to be converted to exceptions.

---

[1] Code for the cryptography algorithms did not have to be converted because classes from the *java.security* and *javax.crypto* packages were used instead.

[2] LOC count includes code that provides the bindings from Java to C.

- The code in the Berkeley DB layer was very complicated and made extensive use of pointers. Also, because of lack of documentation, some portions of the code were converted without understanding the rationale of the implementation (e.g., why is a certain condition testing against a value "10"?).
- Testing required a large effort. A "unit test program" was created for each class because there is no user interface. The system had to be tested against many different databases of different sizes to make sure that all the Berkeley DB code was exercised. A detailed system test plan was created that required generating test programs and using *keytool* – a key/certificate management tool included with the Java Development Kit.
- Adding the write and delete capabilities required just as much effort as converting to 100% Java. Deleting and writing certificates and keys to the keystore required detailed knowledge of exactly how key/data pairs are stored in the Netscape database. Even the Mozilla documentation was unclear; there were errors regarding the structure of some of the record types and some features, albeit promised, were never discussed.

3.1.1 Conversion of the Netscape Keystore Management Code

The first step in the detailed design was to understand the data structures being used, which mostly resembled the different record types in the certificate and key file, and to understand the routines that were using these data structures. While looking at the code, it was clear that although a lot of information is read from the different records, not all of it is actually used.

The second step was to identify the information used in the keystore management process and the routines that worked on these data structures. The third step was to group the routines into logical classes, following the object-oriented concept of encapsulation where data and functions are integrated. Using all this information, we produced the high-level UML (Unified Modeling Language) class diagram for the Netscape keystore layer, shown in Figure 2. Details can be found in Appendix C.

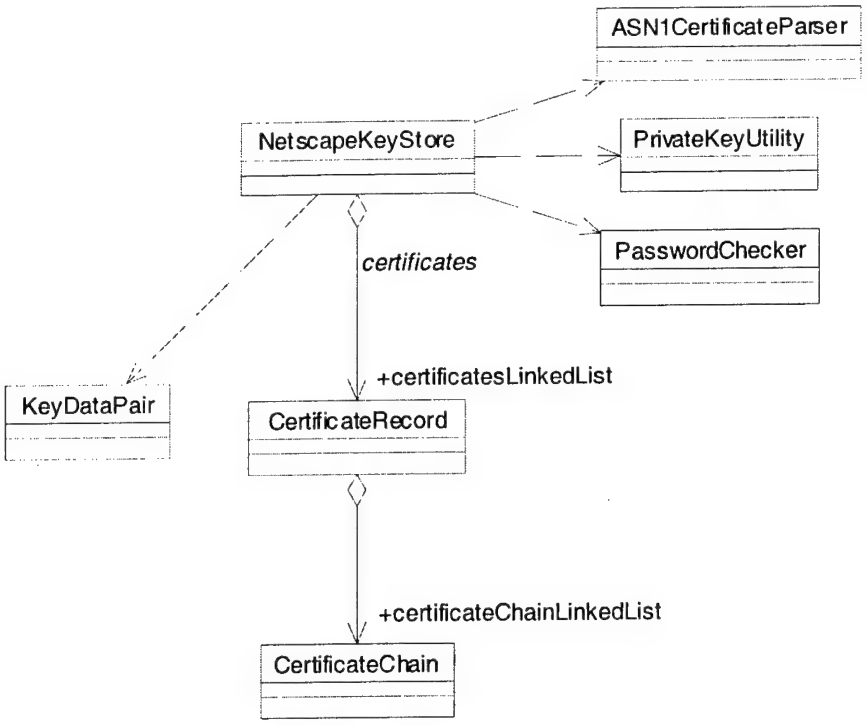


Figure 2: High-Level UML Class Diagram for Netscape Keystore Management

The *NetscapeKeystore* class contains the global Netscape keystore information, such as the certificate and key file names, the global salt,[1] and the hashed password. It has two references to the *BerkeleyDBManager*[2] class that correspond to the certificate and key files. The *KeyDataPair* class is used to pass information in the form of key/data pairs to and from the *NetscapeKeystore* and *BerkeleyDBManager* classes.

The *PasswordChecker* class is used to check the validity of the password provided to the *NetscapeKeystore* class. It interprets the records that store information used in password checking and uses crypto routines to verify the password.

Once the password has been verified, the information from the certificate and key files is stored in the *CertificateRecord* and *CertificateChain* classes. The *CertificateRecord* class contains generic certificate information, such as its alias, and keeps a linked list of all the certificates in its certificate chain using the *CertificateChain* class. These classes do not store the actual information contained in the keystore. Instead, they store the database keys to access the data.

---

1 A salt is a random set of bytes that is concatenated with a password before encrypting it, in order to make an unauthorized decryption harder.

2 The relationship between *Netscape KeyStore Management* and the *BerkeleyDBManager* classes is shown in Figure 4.

Each certificate in the Netscape certificate file contains a reference to an entry in the key file. The *ASN1CertficateParser* class contains the functionality to extract the database key to the entry in the key file from a certificate record. We converted the existing ASN.1 parser code because we were not able to find equivalent existent Java code. We tried to use the Java classes *java.security.cert.X509Certificate* and *java.security.cert.X509Extension* because they contain methods to extract information encoded in a certificate, but we were not able to get access to the specific object identifier (OID) that would identify the piece of information we needed (1.2.840.113549.1.1.1). This OID did not appear either when we called the methods to list all the critical and non-critical extension OIDs. We also tried searching for ASN.1 parsers or Java class generators on the Web, but the work required to integrate them to our code or to generate classes from the ASN.1 encoding would have been harder than converting the existing parser code. It is only used once and is very specific in the search of this particular OID.

Finally, the *PrivateKeyUtility* class extracts and decrypts private key information and builds private key records.

3.1.2 Implementation of the Cryptography Routines

NDBS 2.0 uses any JCE-compliant third-party crypto provider for the algorithms needed to decrypt these private keys, such as HMAC with SHA-1, and Triple DES with CBC and Standard Block Padding. SHA-1 is also used, but it is provided by the *java.security* package.

We first determined if the algorithms used by NDBS 1.0 were based on public crypto standards, and then proceeded to discover if Java provided equivalent algorithms. Risk reduction prototyping was done to determine if the crypto routines in the C code in NDBS 1.0 were equivalent to the routines in the crypto classes provided by JCE, as well as the crypto algorithms provided by a JCE 1.2-compliant crypto provider such as SunJCE.

After verifying that Java provided equivalent crypto routines that could be used for password checking and private key extraction, we produced the class diagram shown in Figure 3. Details can be found in Appendix C.

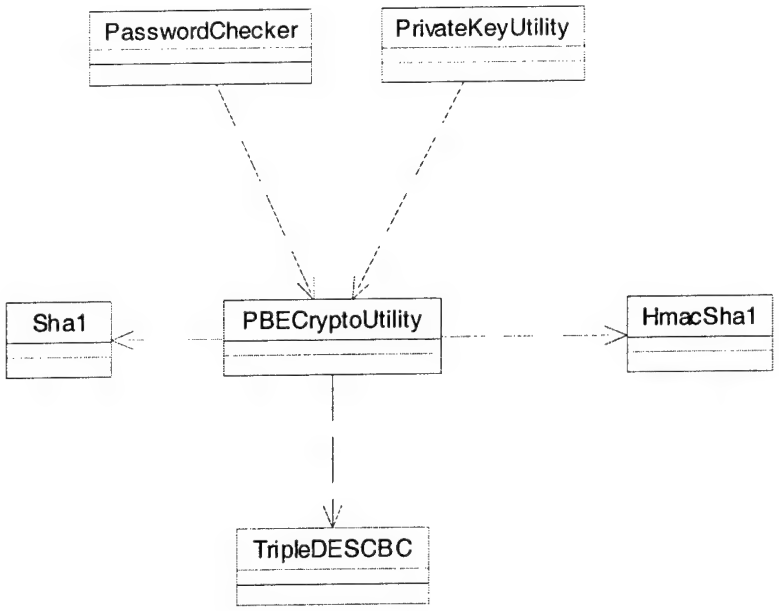


Figure 3: High-Level UML Class Diagram for Cryptography Routines

The *PasswordChecker* class and the *PrivateKeyUtility* class in the keystore management layer use the *PBECryptoUtility* class to perform PBE (Password-Based Encryption) based on the PKCS 12 Standard – PBE with SHA-1 and Triple DES CBC. Unfortunately, Java does not have direct support for PKCS#12; the *PBECryptoUtility* class uses the *Sha1*, *TripleDESCBC*, and *HmacSha1* classes to implement the PBE algorithm. The *Sha1*, *TripleDESCBC*, and *HmacSha1* are façades to the Java Cryptography Extension (JCE) classes that provide the crypto functionality needed by the *PBECryptoUtility* class. This is detailed in Section 4, *Extending the Functionality of NDBS 2.0.*

We encountered several problems when we ran tests using Entrust as one of the crypto providers installed in the JRE_HOME/lib/ext directory. Because the Java Runtime checks the providers in alphabetical order when initializing its crypto classes, it always encountered the *entrust.jar* file first and threw an *IncompatibleClassChangeError* or *ClassNotFound* exception. When we moved the file out of that directory, the tests were successful. After searching on the Internet, we encountered a bulletin board posting that identified a bug in versions of Entrust previous to 4.1. The bug required calling the method *com.entrust.util.Util.initCiphers()* before any of their algorithms can be used. This problem was fixed and confirmed in the documentation for version 4.1 of Entrust. After requesting and installing the new version of Entrust, we had no further problems.

3.1.3 Conversion of the Berkeley DB 1.85 Hash Structure Code to Provide Read Capability

Berkeley DB, which was developed by one of the founders of a company called Sleepycat Software Inc., is the underlying database used by Netscape to store certificates and private keys. While visiting their Web site,[1] we saw that they had a Java API to the database. This was good news because that meant that we could use the API to access the database without having to write code for this. Unfortunately, after downloading the code, we discovered that the API did what NDBS 1.0 does, meaning that it uses JNI to include C code in Java applications. We wrote to Sleepycat to see if a 100% Java API was in development, and if not, whether they would be interested in our effort. Company personnel answered that they had no plans for the 100% Java API, the reason being that it would not offer good performance. To us this seemed very myopic. They also said that they were not interested in our effort because we would be developing the API for version 1.85 of their database and not the current version 3.2.

At that point, we knew we had to convert the hash functionality of the Berkeley DB 1.85 C API that was used by NDBS 1.0 to Java, because NDBS 2.0 had to be 100% Java. The main challenge was the lack of clear documentation of the Berkeley DB source code for the hash structure. We first did some research to figure out what type of hash structure is used by Berkeley DB 1.85. From information on Sleepycat's Web site, we discovered that the hash structure is an extended linear hash. After searching on the Internet for information about extended linear hashing, we found several useful links that helped us understand the hash structure while we were trying to understand the code.

The second challenge was that Berkeley DB 1.85 is written in C, a structured language, which made it harder to redesign the system as an object-oriented system to be coded in Java. Another challenge during the conversion was the extensive use of pointers in the original C code.

In this stage, only the functions open, read sequentially, get a specific record, synchronize, and close were converted. We did not convert the part of the code related to creating new files because it is not needed by NDBS. The synchronize function was not necessary because we were not writing data to files, but we converted it at this point because it was necessary for the write and delete capabilities.

Testing for this part of NDBS 2.0 was difficult because we had to find keystores to exercise all portions of the code. This part was always considered risky because of the lack of documentation—portions of the code were converted without understanding the exact rationale for implementation decisions. The final structure of the Berkeley DB hash functionality is shown in the high-level UML class diagram in Figure 4 on the following page. Details can be found in Appendix C.

---

[1] <http://www.sleepycat.com>

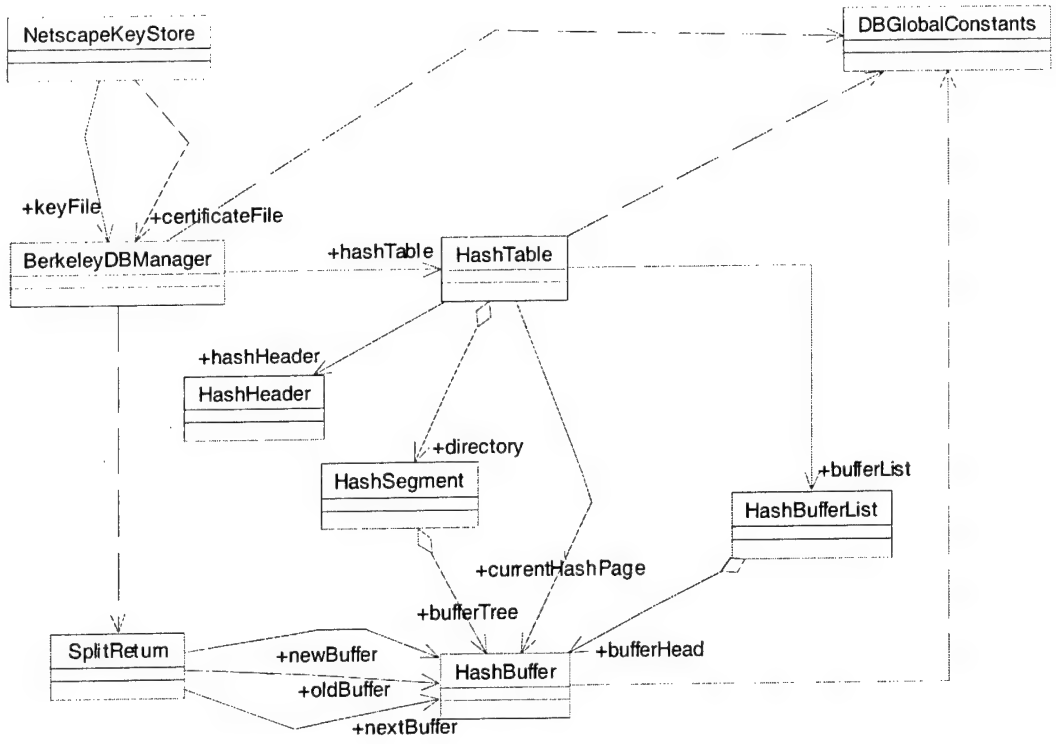


Figure 4: High-Level UML Class Diagram for the Berkeley DB 1.85 Hash Structure

The *HashTable* class maintains the hash table information used in the hash structure. It has a reference to the *HashHeader* class, which contains general information to correctly open and read data from a file. The *HashTable* class has a reference to the *HashSegment* class. This reference is implemented as an array and represents a directory that is used to index the *HashBuffers* so they can be located in the *HashBufferList*. A *HashBuffer* is created every time there is a search for a hash key and the associated buffer is not already in memory. *HashTable* also has a direct reference to the *HashBuffer* class that represents the page being currently accessed. *DBGlobalConstants* defines constants used by the other classes in this layer.

3.1.4 System Integration

The code from the previous layers is integrated into a Java Keystore SPI. A cryptographic service provider that wishes to implement a particular keystore must implement all the abstract methods in the class *java.security.KeyStoreSpi*. The use of this provider is specified in the *java.security* file used by the Java Runtime Environment (JRE).

The design for this part of NDBS is shown in Figure 5. *NDBSKeyStore* implements the methods in the abstract class *java.security.KeyStoreSpi*. *NDBSKeyStore* uses the services of the *GenericKeyStore* class. *GenericKeyStore* is an interface and therefore contains no code for methods, only headers. The methods in the *GenericKeyStore* class correspond to services that are common for all keystore implementations. Common services provided by this class

are, for example, to open a keystore file, get a certificate/private key, write a certificate/private key, delete a certificate/private key, and get a certificate chain.

NetscapeKeyStore and *CSPKeyStore* are specific keystore implementations of *GenericKeyStore* for Netscape and Microsoft CSP respectively. The decision to instantiate either one can be decided at run-time. This is further explained in *Section 4, Extending the Functionality of NDBS 2.0.*

When all the code was finished it was bundled in a package and eventually into a JAR file (Java Archive). This JAR file has to be placed in the JRE_HOME/lib/ext directory and installed as a provider in the *java.security* file for the Java Runtime Engine (JRE) to recognize it as a keystore service provider.[1]

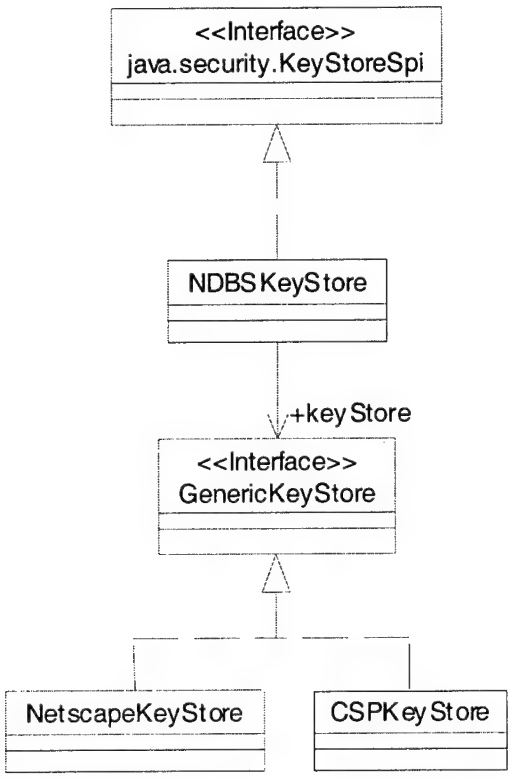


Figure 5: High-Level UML Class Diagram for the Keystore Service Provider Interface

During system testing, we discovered that NDBS 2.0 failed with certain certificates. After tracing the code, we noticed that the Java method *CertificateFactory.generateCertificate()* was throwing a non-declared NullPointerException and returning a null certificate. A close

[1] Details for implementing a provider can be found at <http://java.sun.com/j2se/1.3/docs/guide/security/HowToImplAProvider.html>.

look at the certificates where it failed showed that these were cases where the X.509v3 alternative name extension SubjectAlternativeNameExtension or IssuerAlternativeNameExtension had an empty GeneralNames Sequence. The X.509v3 specification clearly states that this is not allowed.[1] Nevertheless, there are several reports on OpenSSL newsgroups stating that there are Certificate Authorities (CAs) generating certificates with empty general names. This problem was reported to Sun because even though it is not their fault, their code is generating a problem when this situation is encountered.

3.1.5 Addition of the Write and Delete Capabilities

The addition of write and delete capabilities required an effort as large as the conversion to 100% Java. The Java *KeyStoreSpi* class has the following specification for the methods related to writing and deleting keystore entries:

- Writing a certificate to a keystore requires the certificate to be in a variable of type *java.security.Certificate*.
- Writing a private key to a keystore requires the private key to be in a variable of type *java.security.Key* or in a byte array where it has been protected; and the certificate chain that certifies the corresponding public key to be in an array of type *java.security.Certificate,* if the private key is of type *java.security.PrivateKey*.
- Deleting an entry from a file (certificate or private key) requires only an alias.

Given these requirements, the first step was to find available source code that performed this functionality.

Netscape Personal Security Manager (PSM) was the first product we considered. PSM is a Mozilla open source project that is included as part of Netscape 6.x. It is a set of libraries that performs cryptographic operations on behalf of a client, including writing and deleting certificates and private keys. We looked at the source code for PSM and found it overloaded due to its message-based communication between the PSM Client Library and the PSM Daemon, as shown in Figure 6. Following the code was difficult because instead of making direct calls, it embedded the calls in messages.[2]

---

[1] The X.509v3 document RFC (2459) is located at <http://www.ipa.go.jp/security/rfc/RFC2459EN.html>.

[2] More information about PSM can be found at <http://www.mozilla.org/projects/security/pki/psm/>.

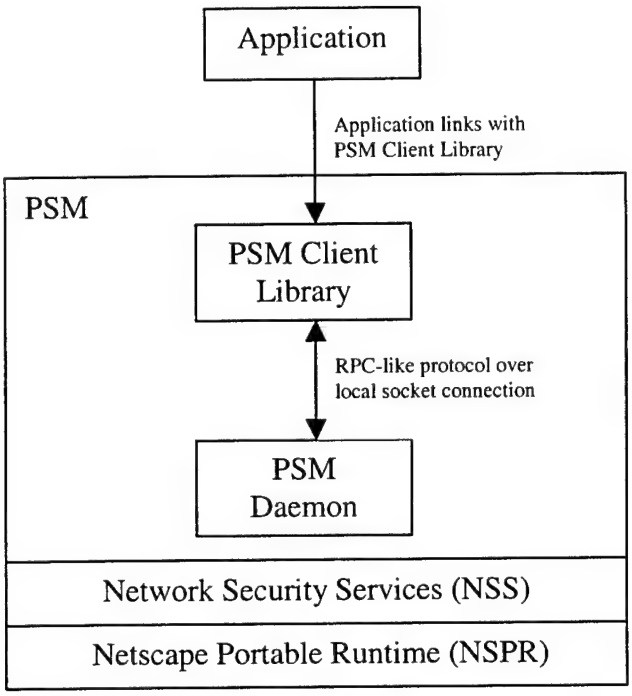


Figure 6: PSM Architecture

In searching for more information about PSM, we found another Mozilla open source project called Certutil, a command-line tool to manage Netscape certificate and key databases. The Certutil source code was much easier to follow and had the functionality that we needed.[1] It was also documented using LXR,[2] a hypertext cross-referencing tool, making it very easy to just click on a link to access the source code for the function.

Looking at the code in detail, it did a lot more than what we actually needed. It creates certificates by loading them from a file that is sent by a CA in response to a certificate request. The information from this file is stored in a number of internal structures that are transferred to the keystore. Certutil was very useful because even though we would not convert the exact code because it didn't match what we needed, it did guide us on what had to be done and identified the functions in the Berkeley DB source code that had to be converted: *hashDelete()* to delete a record from the database and *hashPut()* to write a record to the database.

Writing a certificate to a keystore requires adding three records to the certificate database:

1. a subject record that contains the general certificate information and the database keys to the actual certificate and to the certificates in its certificate chain

[1] More information on Certutil and other Mozilla open source projects can be found at <http://www.mozilla.org/projects/security/pki/nss/tools/index.html>.

[2] More information on LXR can be found at <http://lxr.linux.no/>.

2. a certificate record that corresponds to the first certificate in the chain and contains the actual certificate information
3. either a nickname record, or a MIME profile record, depending on whether the certificate is identified by its nickname or its e-mail address, respectively

Writing a private key to the keystore requires

- writing the private key record to the key database
- creating a certificate, subject, and nickname record for the first certificate in the chain

Deleting a certificate requires

- deleting the three records mentioned when adding a certificate
- deleting the associated private key if it is not the private key for any other certificate in the keystore

Deleting a private key explicitly is not necessary because the KeyStoreSpi only has a method called *engineDeleteEntry()* that deletes the entry associated with the alias given as a parameter. In this case, NDBS 2.0 deletes a private key indirectly when deleting the certificate that corresponds to the given alias.

This information was all obtained by examining the Certutil source code. To become familiar with the database structure, we started by implementing *engineDeleteEntry()* because it seemed simpler. First of all, the internal structure where the certificate information is stored inside *NetscapeKeyStore* was changed so that the database key to the subject record, nickname record, and MIME profile record were saved along with the rest of the certificate information, and set when opening the keystore. With this change, deleting only required calling the *hashDelete()* method in the *BerkeleyDBManager* with each of the database keys stored along with the certificate information.

Writing records to the keystore required researching the information needed to build the key/data pairs for the database. The sources that provided the most information for writing certificates were [Plakosh 99] and a URL on the Mozilla Web site that contained the structure of the certificate database. The information on the Web site was accurate except for the structure of the certificate record database key. In this case, the serial number goes before the issuer distinguished name and not after. Also, the nickname and email do not have at least a length equal to one when they are empty. Fortunately the certificate contained all the information necessary to build the database keys and records. Additional ASN.1 parsing functionality was added to the *ASN1CertificateParser* to be able to extract this information from the certificate [Larmouth 00]. The problem appeared when building the subject record because it requires a *KeyID* for each certificate in the chain and there was no information on what this meant. Searching on the Internet with keywords like "keyid," "Netscape," and "cert7.db," we found that there were many interpretations of what a *KeyId* was: the modulus of the RSA key, the public key hash, the subject issuer distinguished name hash, the SHA-1 fingerprint, and unfortunately none of these were true. After looking at the actual information

stored in the keystore we noticed that these *KeyID*s always had length 20. This was the hint that it was probably a SHA-1 hash of some information inside the certificate, so we started creating SHA-1 hashes for all the information that could be extracted from the certificate, and finally discovered that it was the hash of the Distinguished Encoding Rules (DER)-encoded public key.

For writing the private key record to the key database, the information on <http://www.drh-consultancy.demon.co.uk/key3.html> was sufficient. The private key was encrypted using the database password and a secure random salt as input to the same PBE routines that were used to decrypt the private key [Plakosh 99].

3.2 Architecture of NDBS 2.0

Even though NDBS 2.0 currently supports only Netscape keystores, the architecture is designed to provide extensibility features that will allow it to support other keystores such as Microsoft Crypto Service Provider (CSP keystores).

Based on this architecture, the NDBS system extracts certificate and encrypted key information from the Netscape or Microsoft Crypto Service Provider, and then uses standard algorithms to decrypt the keys, provided the user has entered the correct password for the keystore file. The runtime diagram for NDBS 2.0 is shown in Figure 7.

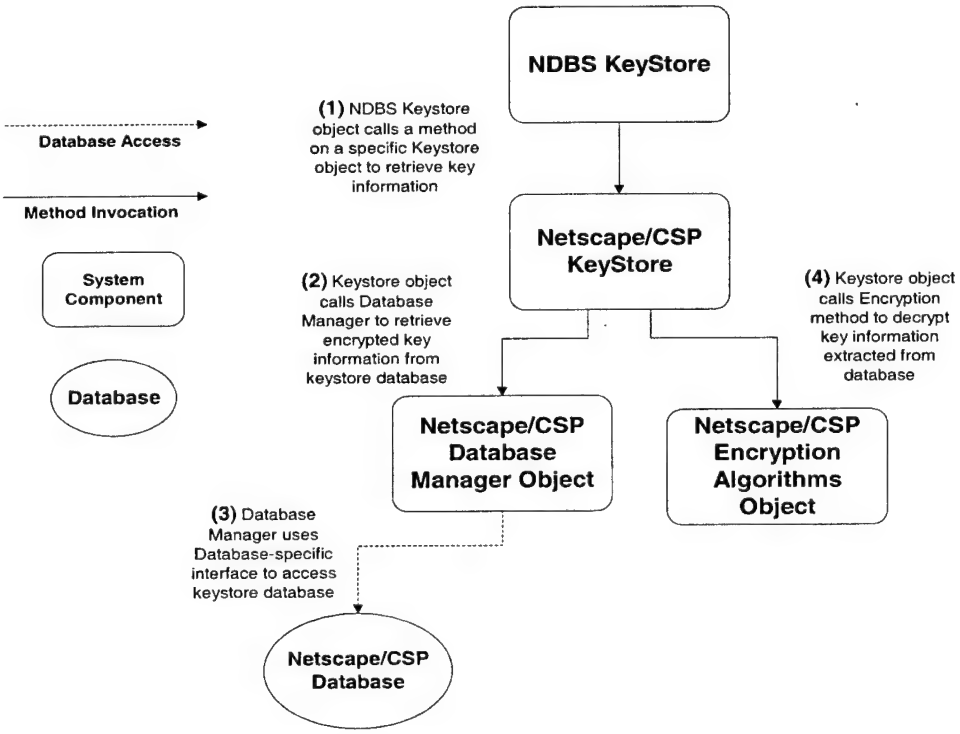


Figure 7: Runtime Architectural Diagram for NDBS 2.0

A layered architecture best represents NDBS 2.0 because the system is organized hierarchically, with each layer providing service to the layer above it. This layered system has several desirable properties, such as providing different levels of abstraction and supporting enhancement. The layered architecture is shown in Figure 8. The decision to instantiate either the Netscape KeyStore or the CSP KeyStore (Cryptographic Service Provider) is decided at run-time.

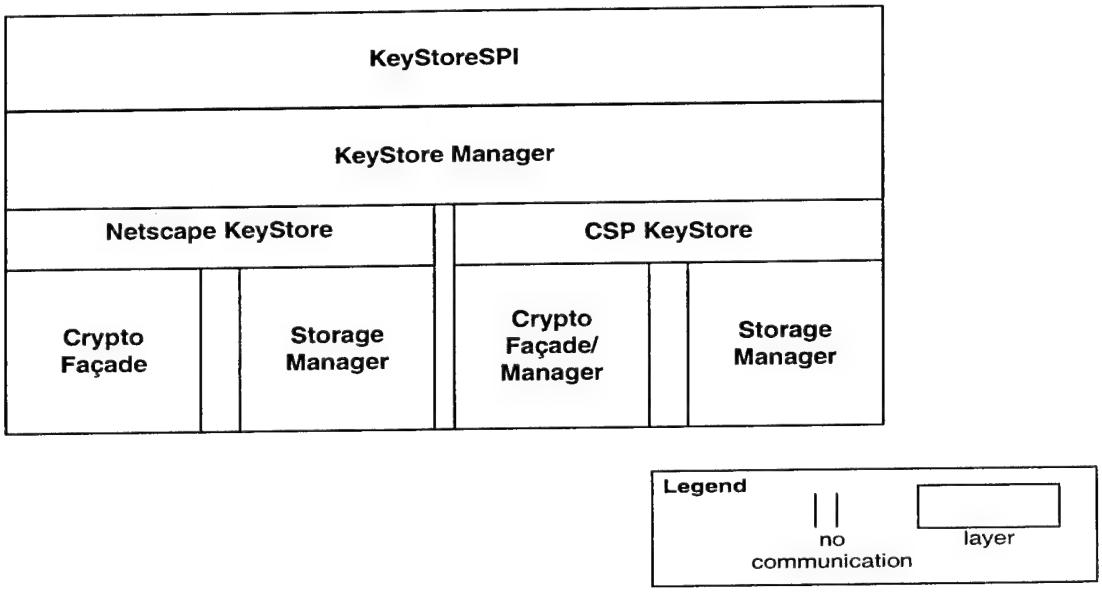


Figure 8: NDBS 2.0 Architecture

The following describes the individual layers:

- KeyStoreSPI – KeyStore Service Provider Interface: This layer implements the Java KeyStore Service Provider Interface (KeyStoreSpi). The use of this provider is specified in the *java.security* file.
- KeyStore Manager: This layer is an interface that specifies the main keystore functionality required by the KeyStoreSPI.
- Netscape KeyStore: This layer implements the functionality required by the KeyStore Manager layer as well as the specific functionality required by Netscape keystores.
- CSP KeyStore: This layer would implement the functionality required by the KeyStore Manager layer as well as the specific functionality required by the Microsoft Cryptographic Service Provider (CSP). This structure is specified for future extensibility of NDBS 2.0 to access Microsoft keystores. More details will be presented in Section 4, *Extending the Functionality of NDBS 2.0.*
- Crypto Façade: This layer contains the crypto functionality used in accessing Netscape keystores. It hides the complexity and possible differences in parameters and output format of the provider crypto algorithms.
- Storage Manager: This layer carries out the operations of accessing information from the databases for Netscape keystores or from a registry or a database for Microsoft keystores.

- Crypto Façade/Manager: This layer will contain the crypto functionality used in accessing Microsoft keystores and will manage the *dll* files that support the crypto services.

Figure 9 provides further insight into the role of each layer and the Java classes that fulfill these roles in NDBS 2.0. Details can be found in Appendix B.

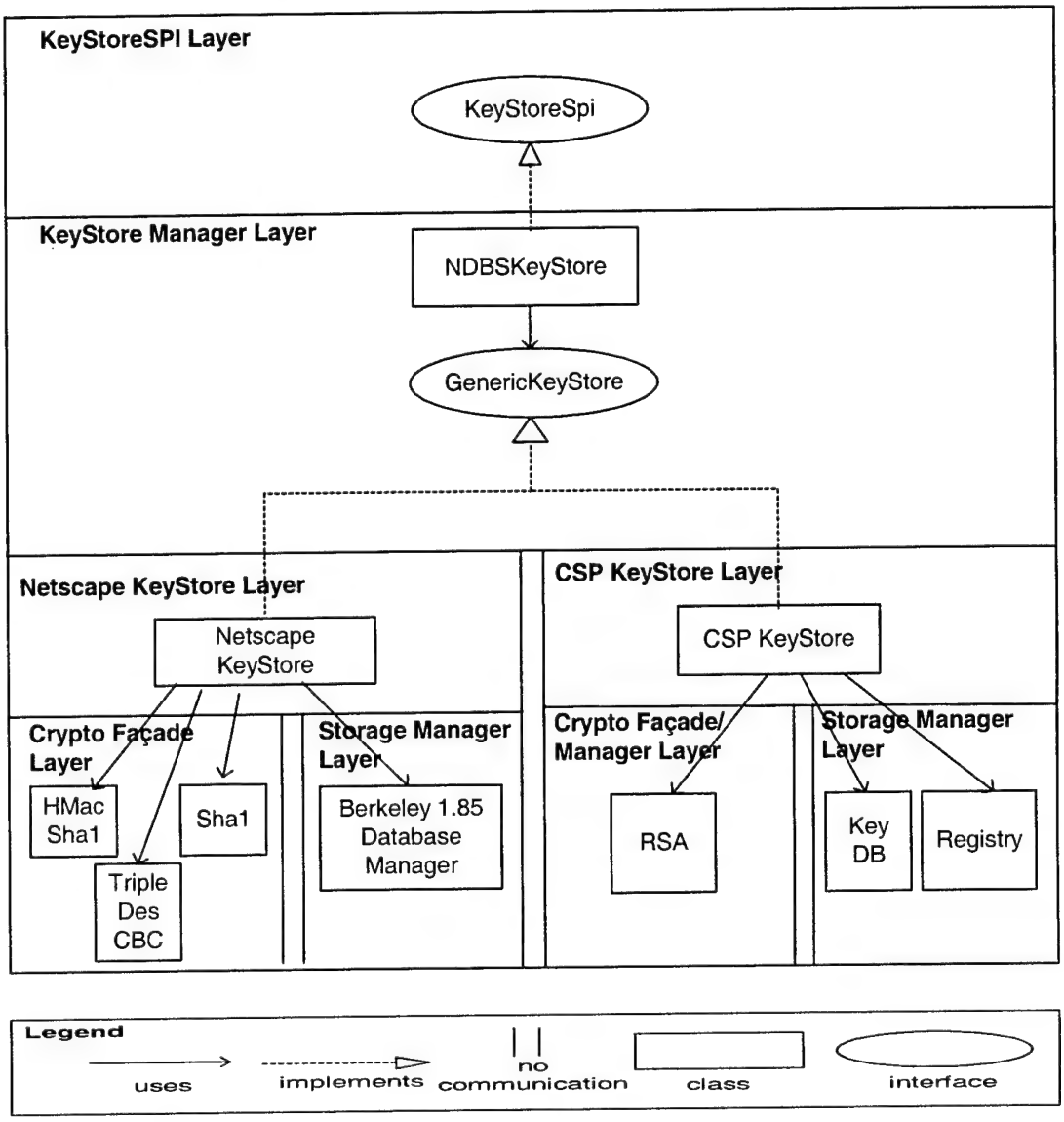


Figure 9: Mapping of Components to the NDBS 2.0 Architecture

The development of the NDBS 2.0 architecture was based primarily on the need to support Microsoft key and certificate databases. Specifically, the team examined the Microsoft Crypto API and Crypto SPI that support the Microsoft security architecture. This examination revealed the following constraints:

- The MS Crypto API does provide the majority of the functions required of the CSP Keystore layer, but does not provide the ability to extract private keys.
- The MS Crypto API and MS Crypto SPI do not support Java.
- The MS Crypto SPI defines standard interfaces and data formats, but everything below the SPI is dependent upon the service provider implementation.

The Microsoft Security Architecture (Appendix A) describes several Cryptographic Service Providers, and each keystore database (whatever format) is encapsulated by the Cryptographic Service Provider. This structure is needed because Microsoft supports different types of keystores for Smartcard, Web Browser, and so on. To extract the data from the keystore, it uses procedural calls to the operating system accessing the Cryptographic Service Providers [Microsoft 00].

4 Extending the Functionality of NDBS 2.0

The following possible scenarios represent extensibility requirements in NDBS 2.0:

- Netscape upgrades the version of Berkeley DB used to store its keystore information.
- NDBS is extended to support Microsoft CryptoAPI keystores.
- The user can choose any crypto provider that is compliant with the Java Cryptography Extension (JCE) and provides algorithms for Triple DES with CBC and HMAC with SHA-1.

NDBS 2.0 provides extensibility features by using design patterns, specifically the Bridge pattern and the Façade pattern [Gamma 95].

The purpose of the Bridge pattern is to separate the interface of a class from its implementation, so that the implementation can be varied or replaced without changing the client code. The use of the Bridge pattern in NDBS 2.0 is shown in Figure 10.

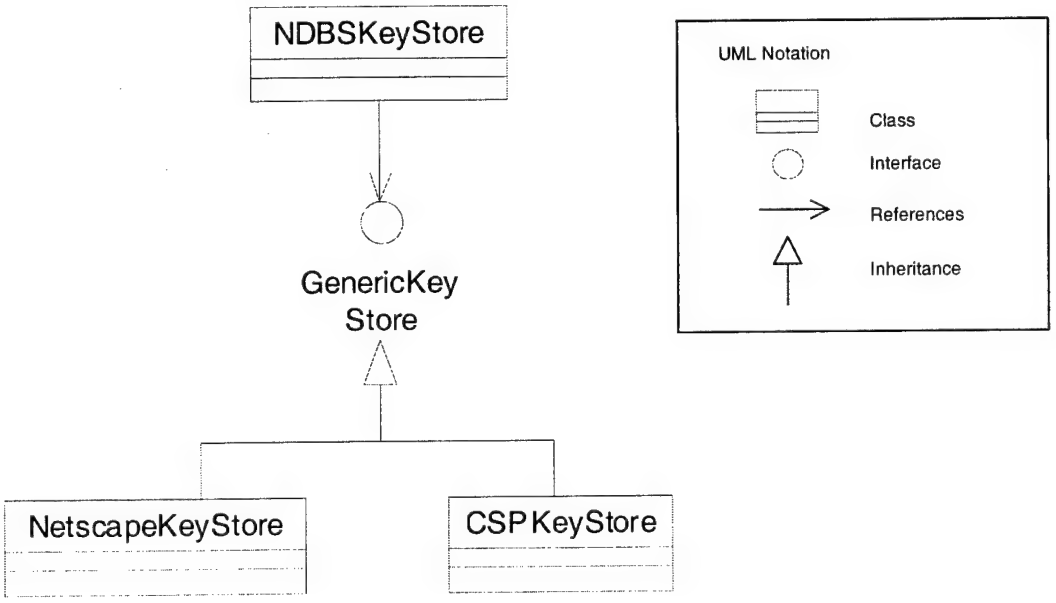


Figure 10: Bridge Pattern in NDBS 2.0

NDBSKeyStore is the class that implements the *KeyStoreSPI* interface and has a reference to an object of type *GenericKeyStore*. Because *NetscapeKeyStore* and *CSPKeyStore* implement *GenericKeyStore*, the decision to instantiate either one can be decided at run-time. The code in *NDBSKeyStore* for this purpose is shown in Figure 11. It assumes that *inputStream* has been set to the configuration file and that this file contains a line that says, for example,

KEYSTORE_CLASS_NAME = edu.cmu.sei.cbs.ndbs.NetscapeKeyStore. This would set the class that implements *GenericKeystore* to *edu.cmu.sei.cbs.ndbs.NetscapeKeystore*.

```
edu.cmu.sei.cbs.ndbs.GenericKeyStore keyStore = null;

java.util.Properties p = new Properties();
p.load(inputStream);

keystoreClassName =
      p.getProperty("KEYSTORE_CLASS_NAME");

try {
   Object classObj =
      Class.forName(keystoreClassName).newInstance();
   keyStore = (GenericKeyStore) classObj;
} catch(Exception e) {
   throw new java.io.IOException("Invalid value for
      KEYSTORE_CLASS_NAME.");
}
```

Figure 11: Code in NDBSKeystore.java that Implements the Bridge Pattern

We chose to use the bridge pattern because it can be adapted easily if NDBS is extended to support Microsoft CSP Keystores. Only these changes would have to be made to support the extension:

- Implement the methods that are described in the *GenericKeystore* interface in *CSPKeyStore*.
- Implement specific methods for the CSP keystore manipulation.
- Change the value of *KEYSTORE_CLASS_NAME* in the configuration file to *CSPKeystore*.

The key architectural tradeoff to be considered is the extent to which the NDBS 2.0 architecture should utilize the MS Crypto API and SPI. The MS Crypto SPI does provide the ability to encrypt and decrypt messages, as well as create, store, and retrieve certificates. However, the MS Crypto SPI does not provide the ability to obtain the private keys. This limitation could be addressed in one of two ways:

1. Wrap the existing MS Crypto SPI and supplement the existing functionality to support the full NDBS 1.0 functionality. This would reduce the development time, but sacrifice the 100% Java requirement. The biggest benefit of this option is that the CSP-dependent certificate database formats are hidden from the NDBS architecture, so only the private key storage structure will differ across CSPs. This option should not affect portability because the Netscape Keystore layer would operate in all non-Microsoft platforms.

2. Ignore the existing MS Crypto SPI and create a Java implementation of CSP Keystore layer that is specific to each CSP. This option adheres to the 100% Java rule, but adds a risk to the architecture because it exposes the CSP implementation dependency to the NDBS architecture. This option should not affect portability because the Netscape Keystore layer would operate in all non-Microsoft platforms.

If Netscape were to upgrade the version of Berkeley DB used to manage its keystores from 1.85 to 3.2, the same bridge pattern, at the same level, would be used. It wouldn't be necessary to add an additional bridge for the different implementation of Netscape keystores because version 2.X of Berkeley DB is interface compatible with 1.85 but not implementation compatible, and version 3.X changed part of its interface as well as the database structure. This means that the interface and the underlying keystore implementation is different, which means that code reuse would probably be small. In this case, an additional class that implements *GenericKeyStore* would be added, as well as an additional possible value for *KEYSTORE_CLASS_NAME*, as indicated in the previous example.

The Façade pattern supports different crypto providers, as well as hiding the complexity and possible differences in parameters and output format of the provider crypto algorithms. The purpose of the Façade pattern is to decrease the complexity of a subsystem by providing a simplified interface to this subsystem [Gamma 95].

NDBS uses three crypto algorithms: SHA-1, HMAC with SHA-1, and Triple DES with CBC and Standard Block Padding. SHA-1 is provided by the *java.security* package, but the other two have to be provided by a crypto provider. Each of these has a façade class that hides the implementation details due to differences in crypto providers. An example for the HMAC with SHA-1 algorithm façade class is shown in Figure 12. The *HMACSha1* façade class implements the methods that would be called if the methods in the *javax.crypto.Mac* class were used directly.

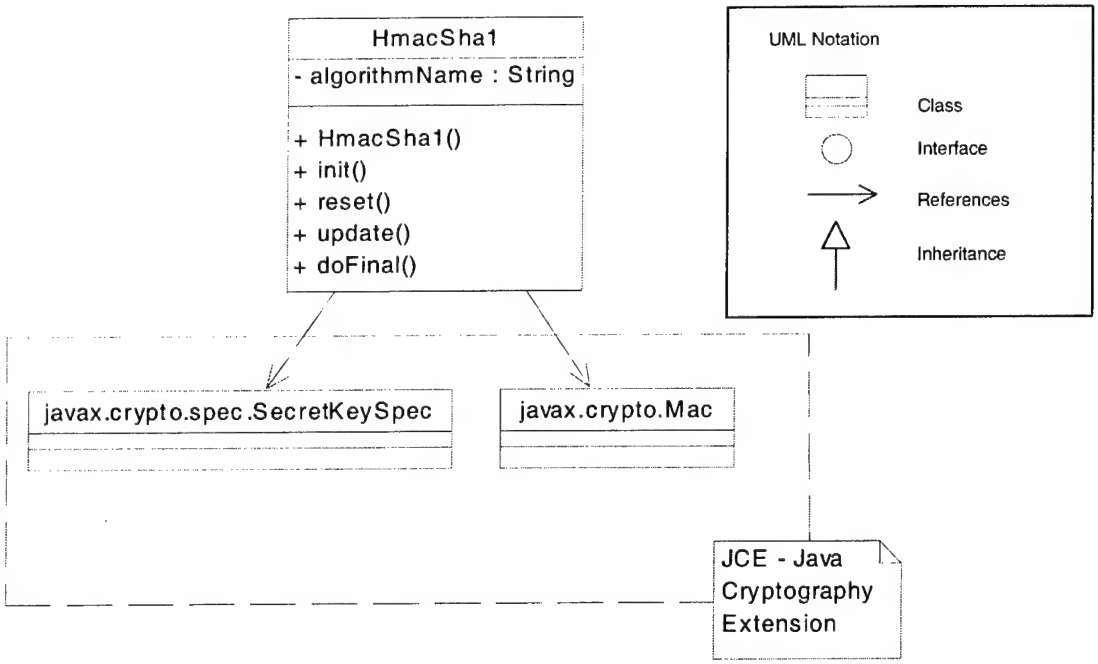


Figure 12: Façade Pattern in NDBS 2.0 for HMAC SHA-1

The advantages of the Façade pattern in this context are

- The Java classes javax.crypto.spec.SecretKeySpec and javax.crypto.Mac are hidden, therefore providing a cleaner implementation because the keystore code only knows the HmacSha1 class and its methods.
- The selection of the algorithm name is hidden in this class, with the added advantage of being able to specify a Java property if the default algorithm names do not match the one in the user's crypto provider.
- The details of converting the key into a SecretKeySpec that is accepted by the algorithm are hidden.
- If the code included in HMAC SHA-1 would happen to be incompatible with the implementation of a given crypto provider, changes could be done in this class without affecting the rest of the code.

Sample code is shown in Figure 13. The structure and code for the classes that implement the Triple DES with CBC as well as the Standard Block Padding and SHA-1 façades are very similar.

A possible scenario that is not supported in NDBS 2.0 is if Netscape decided to use a different encryption algorithm for its private keys (the algorithm OID – Object Identifier). This would require replacing the implementation of the *PBECryptoUtility* class to reflect the new algorithm. If this is the only change, then code modifications would be limited to this class.

```
import javax.crypto.*;
import java.security.*;
import javax.crypto.spec.*;

public class HmacSha1 {

    // HMAC with SHA-1 algorithm name used by crypto provider.
    private String algorithmName = null;

    // Secret key to be used in HMAC with SHA-1 algorithm.
    private javax.crypto.spec.SecretKeySpec secretKey = null;

    // Instance of Mac used for the creation of the message
    // authentication code.
    private javax.crypto.Mac mac = null;

    public HmacSha1() throws java.security.GeneralSecurityException
    {
       // First tries to create an instance of an HMAC object
       // assuming the user has set the java property
       // ndbs.edu.cmu.sei.cbs.hmacAlgorithmName
       String algorithm =
       System.getProperty("edu.cmu.sei.cbs.ndbs.hmacAlgorithmName");
       if (algorithm != null) {
          try {
              algorithmName = algorithm;
             mac = Mac.getInstance(algorithm);
          }
          catch (java.security.NoSuchAlgorithmException e1) {
             throw new java.security.GeneralSecurityException("The
             selected crypto provider does not support the " +
             algorithmName + " algorithm name.");
          }
       }
       else {
          // Tries to create an instance of an HMAC object using
          // HmacSha1 as the algorithm name (SunJCE)
          try {
             algorithmName = "HmacSha1";
             mac = Mac.getInstance(algorithmName);
          }
          catch (java.security.NoSuchAlgorithmException e2) {
             // Tries to create an instance of an HMAC object
             // using HMACwithSHA1 as the algorithm name (JCSI
             // provider)
             try {
                algorithmName = "HMACwithSHA1";
                mac = Mac.getInstance(algorithmName);
             }
             catch (java.security.NoSuchAlgorithmException e3) {
                throw new
                   java.security.GeneralSecurityException("The
                   selected crypto provider does not support any
                   of the default algorithm names.");
             }
          }
       }//if
    }//Constructor
```

Figure 13: Java Sample Code for Implementing the Façade Pattern for the HMAC with SHA-1 Algorithm in NDBS (Part 1 of 2)

```
    public void init(byte[] key) throws
           java.security.GeneralSecurityException
    {
        // Converts the key into an HMAC SHA-1 key
        secretKey = new SecretKeySpec(key, algorithmName);

        try {
            mac.init(secretKey);
        }
        catch (java.security.InvalidKeyException e) {
            throw new java.security.GeneralSecurityException("The
            secret key passed is invalid.");
        }
    }

    public void reset()
    {
        mac.reset();
    }

    public void update(byte[] plainText)
    {
        mac.update( plainText );
    }

    public byte[] doFinal()
    {
        return (mac.doFinal());
    }

    public void setAlgorithmName(String name)
    {
        algorithmName = name;
    }
}
```

Figure 14: Java Sample Code for Implementing the Façade Pattern for the HMAC with SHA-1 Algorithm in NDBS (Part 2 of 2)

5 Summary

The outcome of the NDBS 2.0 project was an enhanced 100% Java version of NDBS. There were challenges in the process due to our unfamiliarity with the cryptography domain, and the differences between a structured language such as C and an object-oriented language such as Java. NDBS 2.0 was a complete re-design of NDBS 1.0, but all its source code was converted and used, with the exception of the crypto algorithms that are provided by any JCE 1.2-compliant crypto provider.

Adding write and delete capabilities was an effort equal to converting to 100% Java. It took longer than expected because the source code for the underlying database manager and the Netscape keystore structure are not fully documented and some of the code was very difficult to debug.

There are several lessons learned from this experience that can be applied to other code conversion efforts.

1. Become familiar with the domain.

   When we started this project, our greatest concern was that we were not familiar with the cryptography domain and all the standards used by Netscape to store information in its databases. Techniques we used to gain familiarity were building a glossary, reading, searching for information on the Internet, and conducting question and answer sessions with experts. Even though now we agree that we spent way too much time in this stage, we also agree that the project would have been impossible without the knowledge we gained during those first months.

2. Build prototypes.

   The code to be converted had very little documentation. We identified parts of the project that were critical because of their complexity and uncertainty of its feasibility in Java: access to the Netscape database from Java, use of crypto algorithms, and the ASN.1 parser. We built prototypes for each and determined that the first two were not a problem and that we had to either convert or rewrite an ASN.1 parser because Java did not provide this functionality. The time spent in this effort paid off in the end because we became comfortable with parts of the code we had to convert, we didn't have to write the crypto algorithm code, and the experience in ASN.1 made the addition of the write capability much easier.

3. Design, design, design.

   NDBS 2.0 required a total re-design because we were converting from a structured language to an object-oriented language and were adding the extensibility requirement. The design also had to incorporate the addition of the write and delete capabilities in the second stage. We developed an architecture for NDBS 2.0 and used a tailored short version of the Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) to verify the architecture against requirements and to evaluate design tradeoffs [Kazman 00].

   For the detailed design we used Rational Rose and generated the Java code skeletons for each class with all the documentation, including preconditions and postconditions. Not only did this save coding time, but it also served as the project specification that was used by the team during implementation.

4. Search for third-party code and use the Internet as a source for information.

   This project used Berkeley DBM, PSM, and Certutil third-party code. The code was either converted or used as a guideline for the NDBS 2.0 implementation. The crypto algorithms did not have to be written nor converted because any third-party JCE 1.2-compliant crypto provider has this functionality. Code for ASN.1 parsers is also available on the Internet, but unfortunately, either it did not fit our requirements or it was not available without a fee. This would have also been a problem for distribution as a part of NDBS 2.0. The third-party code was tested by comparing its output with output from other tools or from the C code.

   The Internet was a valuable source of information, not only to locate the above source code, but also to find information on general cryptography, Java cryptography, and the Netscape database structure. Specialized bulletin boards where problems, product information, and troubleshooting issues were posted turned out to be a very valuable resource in solving our own problems.

Finally, going "beyond the black box" would further support the observation made in the SEI Technical Note documenting the NDBS 1.0 experience [Plakosh 99]: *" ... building systems from a commercial software product often requires more, rather than less, technical sophistication on the part of the software developers ...*", and we would add *... even if all the information and source code is available.*

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University

References

[Gamma 95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[Kazman 00] Kazman, R.; Klein, M.; & Clements, P. *ATAM: Method for Architecture Evaluation Tradeoff Analysis Method.* (CMU/SEI-2000-TR-004, ADA382629) Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>

[Larmouth 00] Larmouth, J. *ASN.1 Complete.* San Francisco, CA: Morgan Kaufmann Publishers, 2000.

[Microsoft 00] Microsoft CSP Architectural Overview. Available on the Web at <URL: http://msdn.microsoft.com/library/psdk/cryptcsp/aboutcsp_5rg7.htm> (2000).

[Plakosh 99] Plakosh, D.; Hissam, S.; & Wallnau, K. *Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software.* (CMU/SEI-99-TN-010 Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. WWW: <URL: http://www.sei.cmu.edu/publications/documents/99.reports/99tn010/99tn010abstract.html> (1999).

Appendix A: Acronyms and Terms

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CA	Certificate Authority
CBC	Cipher Block Chaining
CMU	Carnegie Mellon University
CSP	Cryptography Service Provider
DB	Database
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DLL	Dynamic Link Library
HMAC	Keyed-Hash Message Authentication Code
JAR	Java Archive
JCE	Java Cryptography Extension
JDK	Java Development Kit
JNI	Java Native Interface
JRE	Java Runtime Engine
LOC	Lines of Code
MAC	Message Authentication Code

MIME	Multipurpose Internet Mail Extensions
MSE	Master of Software Engineering
NDBS	Netscape Database Keystore
NSS	Network Security Services
OID	Object Identifier
PBE	Password-Based Encryption
PKCS	Public Key Cryptography Standards
PKI	Public Key Infrastructure
PSM	Personal Security Manager
RSA	Name given to the crypto system developed by Ronald Rivest, Adi Shamir, and Leonard Adieman
SHA-1	Secure Hash Algorithm
SPI	Service Provider Interface
SSL	Secure Socket Layer
X.509v3	International Telecommunications Union (ITU) Standard for Certificates
UML	Unified Modeling Language

Appendix B: Microsoft Security Architecture

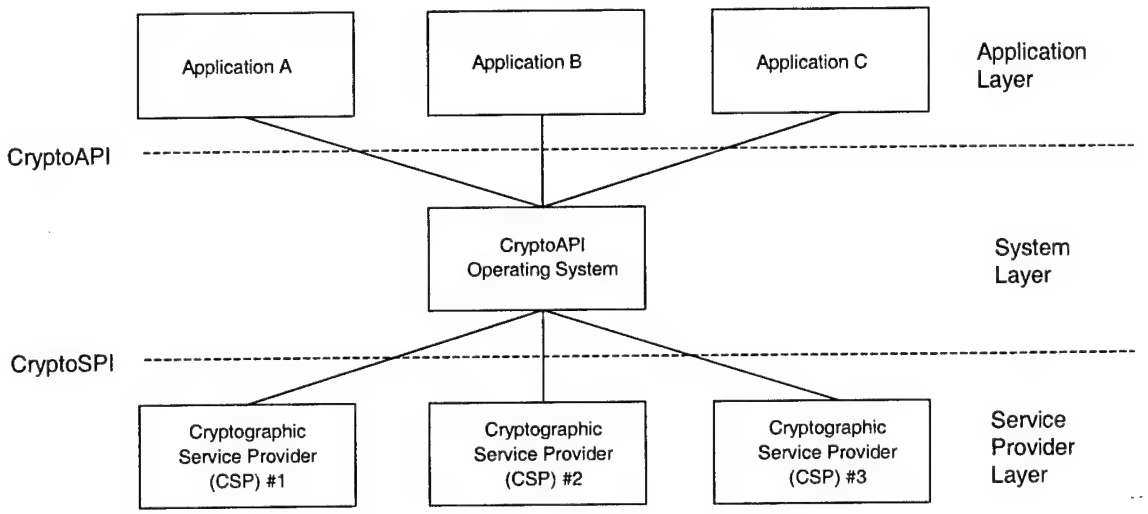


Figure 15: Microsoft Security Architecture

Appendix C: NDBS 2.0 UML Class Diagrams

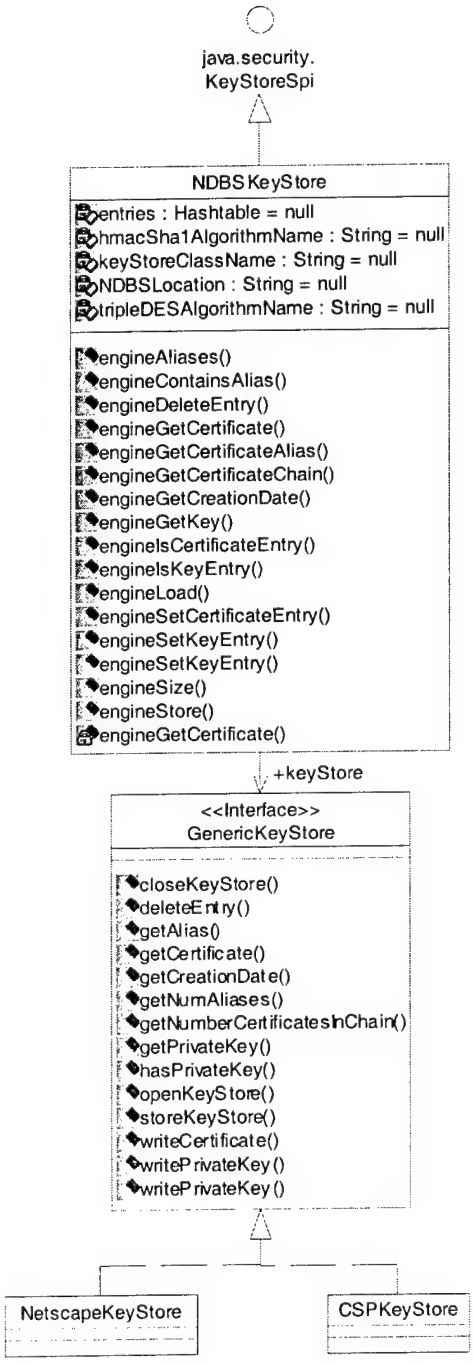


Figure 16: UML Class Diagram for the Keystore Management Layer

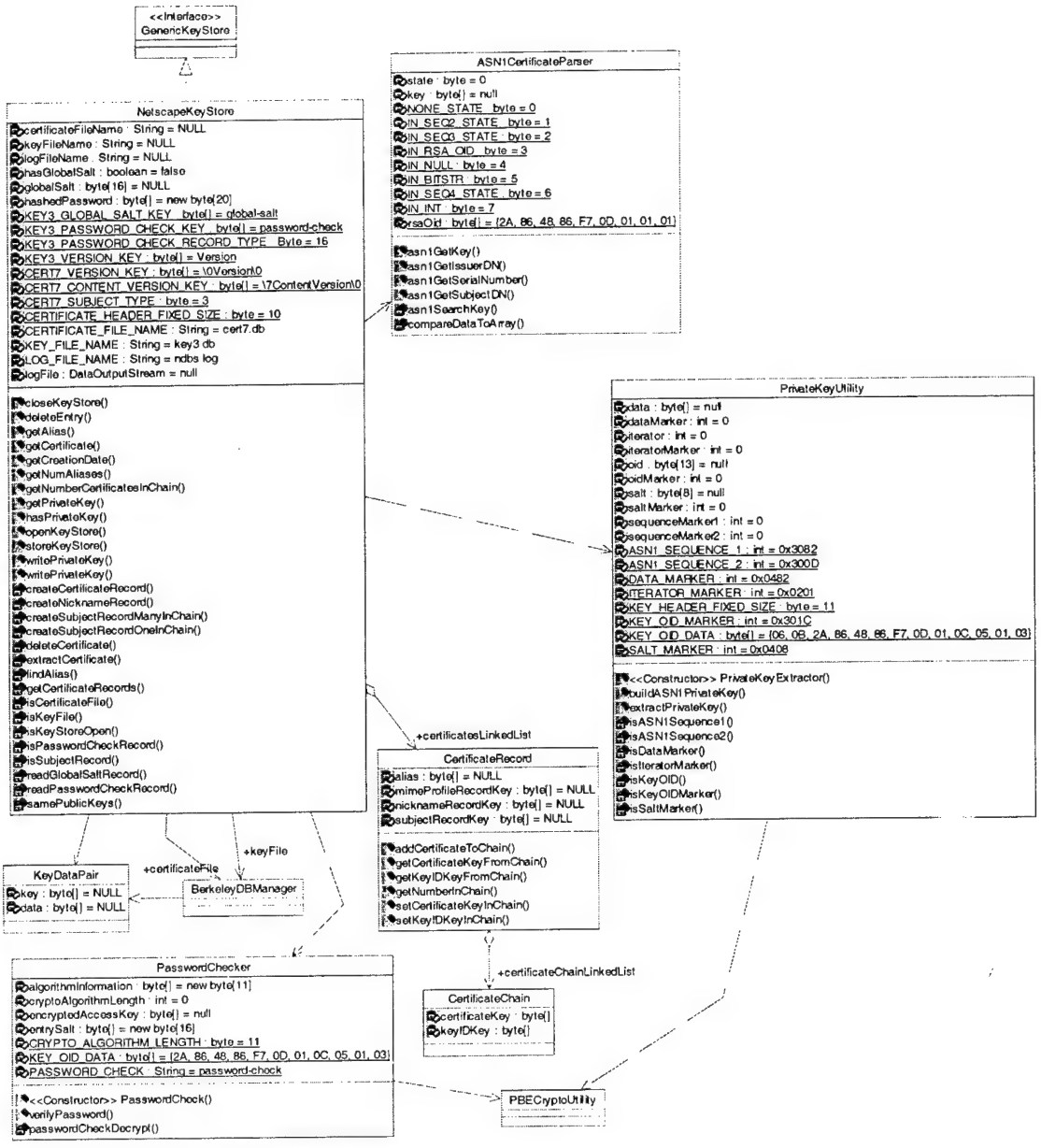


Figure 17: UML Class Diagram for the Netscape Keystore Layer

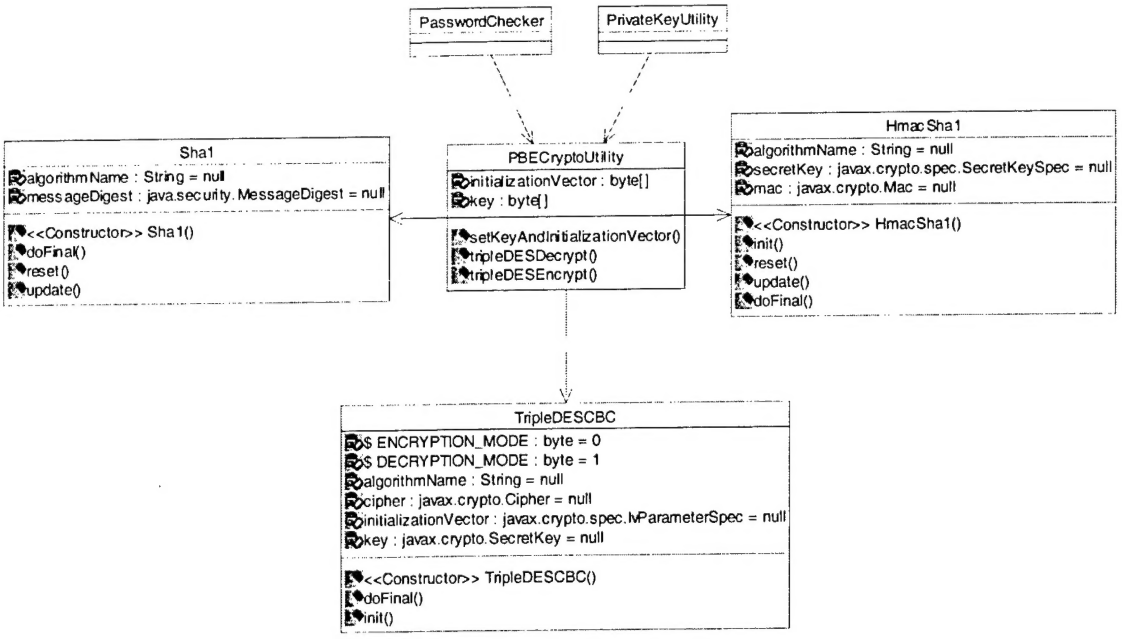


Figure 18: UML Class Diagram for the Netscape Crypto Façade Layer

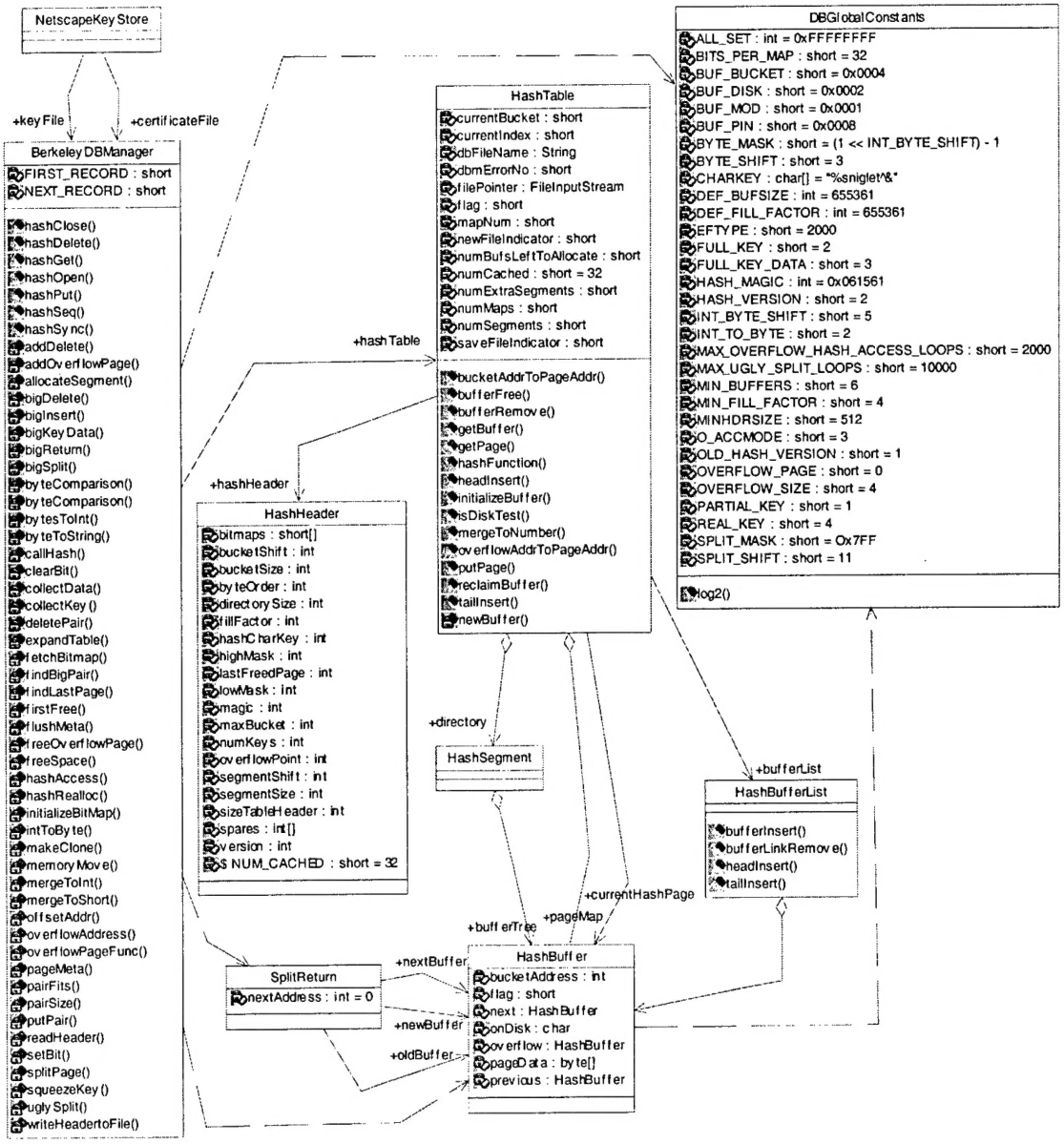


Figure 19: UML Class Diagram for the Berkeley DB Access Layer

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2001	Final

4. TITLE AND SUBTITLE
Beyond the Black Box: A Case Study in C to Java Conversion and Product Extensibility

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Pisey Huy, Grace A. Lewis, Ming-hsun Liu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI 2001-TN-017

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This case study describes the experience of converting and enhancing NDBS 1.0 (Netscape Database Keystore), a programmatic library to extract private keys and digital certificates from a Netscape database written in C and Java. The result of this work is NDBS 2.0, a 100% Java version of NDBS 1.0 designed to support other keystores easily. NDBS 2.0 also includes write and delete capabilities, features that were not present in NDBS 1.0. The case study describes the experience of the conversion and development process, difficulties, and lessons learned.

14. SUBJECT TERMS	15. NUMBER OF PAGES
data conversion and enhancement	49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL